System design: Instagram

Suresh Podeti
7 min read · Oct 24, 2023

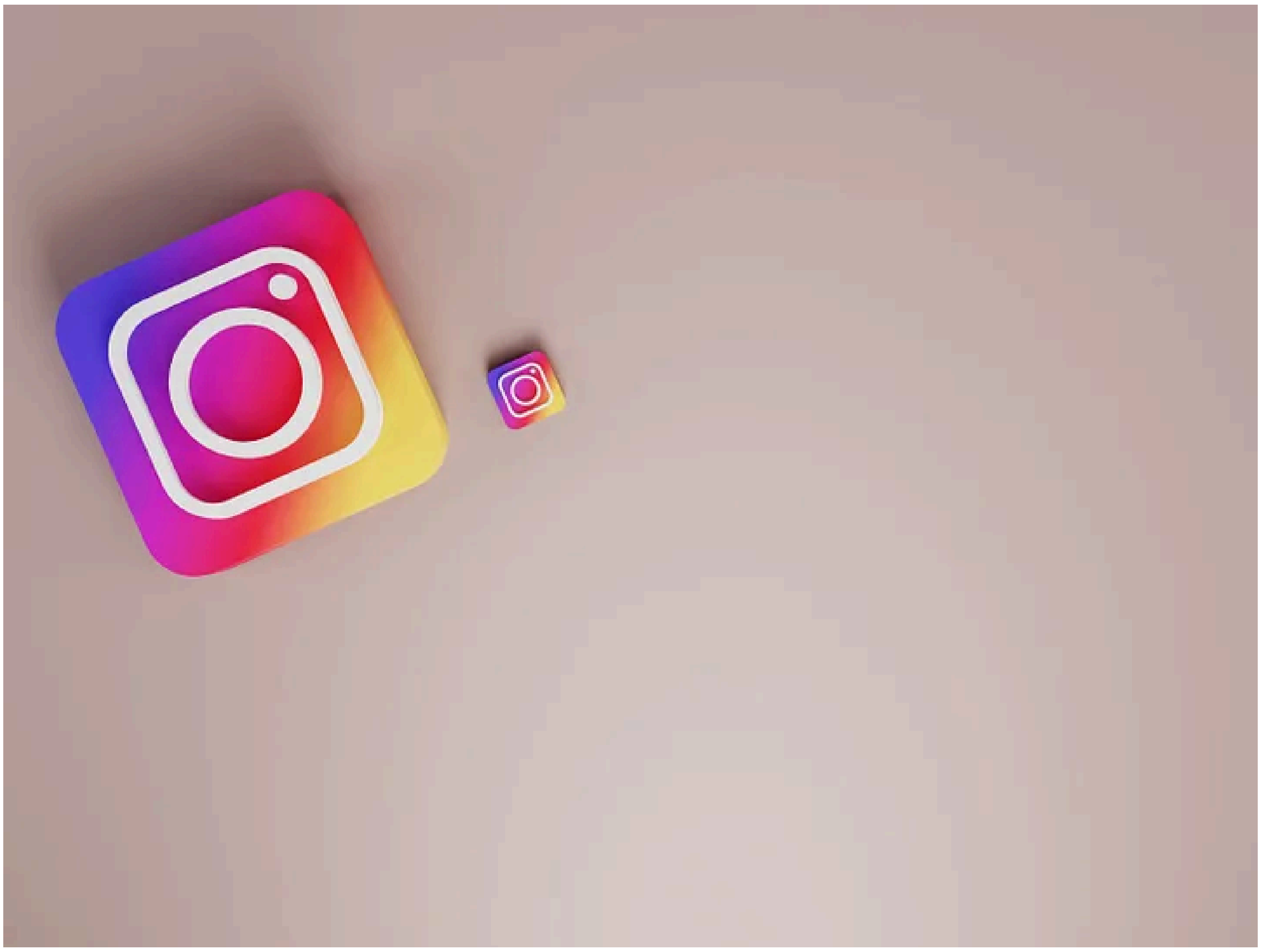


Photo by Deeksha Pahariya on Unsplash

Introduction

Instagram is a **free social app** for **sharing photos and videos with captions.** Posts can be organized using **hashtags** and geotags, making them searchable. Posts are visible to followers and the public if tagged. Users can set profiles to private for restricted access to followers.

Requirements

Functional

- **Post photos and videos:** The users can post photos and videos on Instagram.
- **Follow and unfollow users:** The users can follow and unfollow other users on Instagram.
- **Like or dislike posts:** The users can like or dislike posts of the accounts they follow.
- **Search photos and videos:** The users can search photos and videos based on captions and location.
- **Generate news feed:** The users can view the news feed consisting of the photos and videos (in chronological order) from all the users they follow. Users can also view suggested and promoted photos in their news feed.

Non-functional

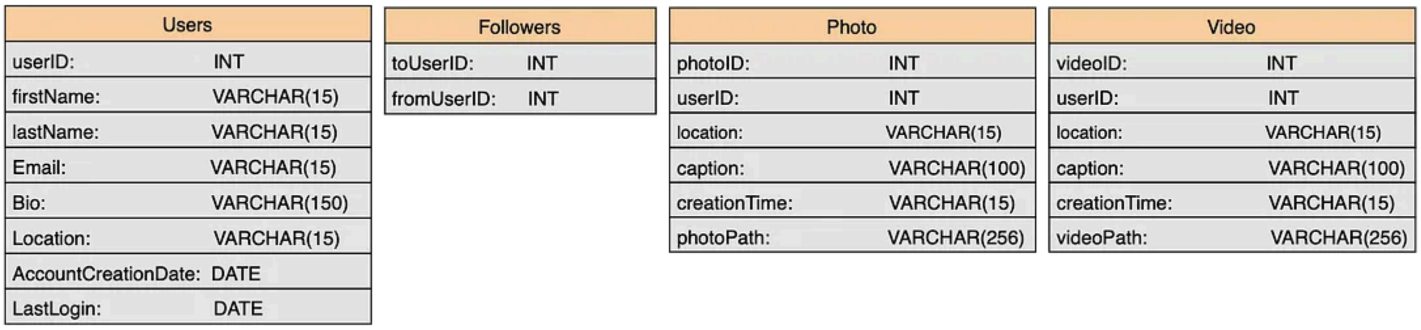
- **Scalability:** The system should be scalable to **handle millions of users** in terms of computational resources and storage.
- **Latency:** The latency to generate a news feed should be low.
- **Availability:** The system should be highly available.
- **Durability** Any uploaded content (photos and videos) should never get lost.
- **Consistency:** We can **compromise a little on consistency**. It is acceptable if the content (photos or videos) takes time to show in followers' feeds located in a distant region.
- **Reliability:** The system must be able to tolerate hardware and software failures.

Storage schema

Entities

- **Users:** This stores all user-related data such as ID, name, email, bio, location, date of account creation, time of the last login, and so on.
- **Followers:** This stores the relations of users. In Instagram, we have a **unidirectional relationship**, for example, if user A accepts a follow request from user B, user B can view user A's post, but vice versa is not valid.

- **Photos:** This stores all photo-related information such as ID, location, caption, time of creation, and so on. We also need to keep the user ID to determine which photo belongs to which user. The user ID is a foreign key from the users table.
- **Videos:** This stores all video-related information such as ID, location, caption, time of creation, and so on. We also need to keep the user ID to determine which video belongs to which user. The user ID is a foreign key from the users table.



Users	
userID:	INT
firstName:	VARCHAR(15)
lastName:	VARCHAR(15)
Email:	VARCHAR(15)
Bio:	VARCHAR(150)
Location:	VARCHAR(15)
AccountCreationDate:	DATE
LastLogin:	DATE

Followers	
toUserID:	INT
fromUserID:	INT

Photo	
photoID:	INT
userID:	INT
location:	VARCHAR(15)
caption:	VARCHAR(100)
creationTime:	VARCHAR(15)
photoPath:	VARCHAR(256)

Video	
videoID:	INT
userID:	INT
location:	VARCHAR(15)
caption:	VARCHAR(100)
creationTime:	VARCHAR(15)
videoPath:	VARCHAR(256)

Fig 1.0: The Data model of Instagram

Relational or non-relational database

It is essential to choose the right kind of database for our Instagram system, but which is the right choice — SQL or NoSQL? Our data is inherently relational, and we need an order for the data (posts should appear in chronological order) and no data loss even in case of failures (data durability). Moreover, in our case, we would benefit from relational queries like fetching the followers or images based on a user ID. Hence, SQL-based databases fulfill these requirements.

So, we'll opt for a **relational database** and store our relevant data in that database.

High level design

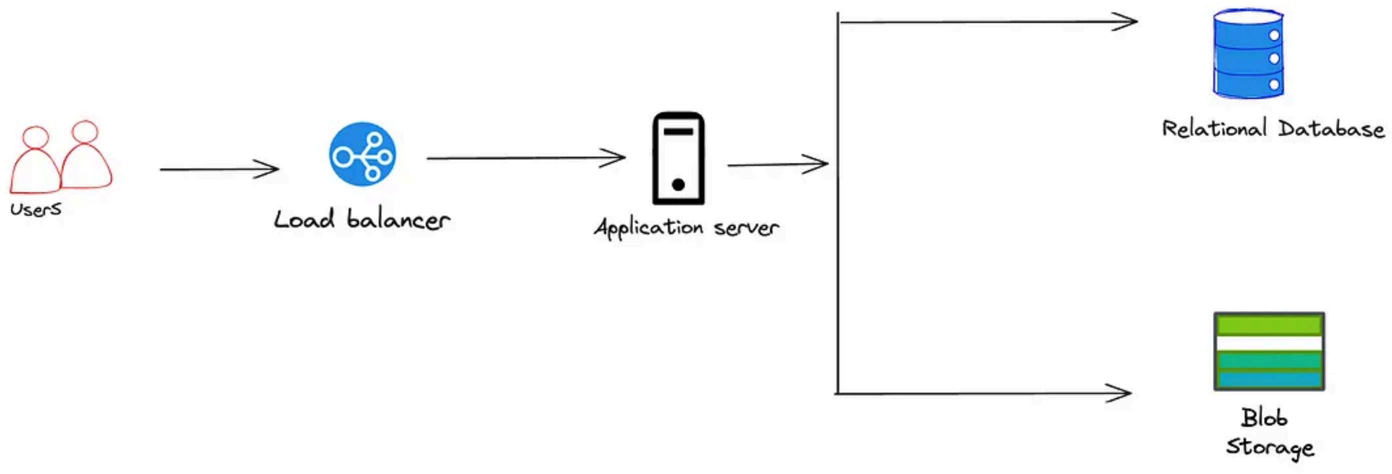


Fig 2.0: High level design

- **Load balancer**: To balance the load of the requests from the end-users.
- **Application servers**: To host our service to the end-users.
- **Relational database**: To store our data.
- **Blob storage**: To store the photos and videos uploaded by the users.

Detailed design

Upload, view, and search a photo

The client requests to upload the photo, load balancer passes the request to any of the application servers, which adds an entry to the database. An update that the photo is stored successfully is sent to the user. If an error is encountered, the user is communicated about it as well.

The photo viewing process is also similar to the above-mentioned flow. The client requests to view a photo, and an appropriate photo that matches the request is fetched from the database and shown to the user. **The client can also provide a keyword to search for a specific image.**

The **read requests are more than write requests** and it takes time to upload the content in the system. It is efficient if we **separat**e the write (uploads) and read services. The multiple services operated by many servers handle the relevant requests. The read service performs the tasks of fetching the required content for the user, while the write service helps upload content to the system.

We also need to **cache the data** to handle millions of reads. It improves the user experience by making the fetching process fast. We'll also opt for **lazy loading**, which minimizes the client's waiting time. It allows us to load the content when the user scrolls and therefore **save the bandwidth** and focus on loading the content the user is currently viewing. It improves the latency to view or search a particular photo or video on Instagram.

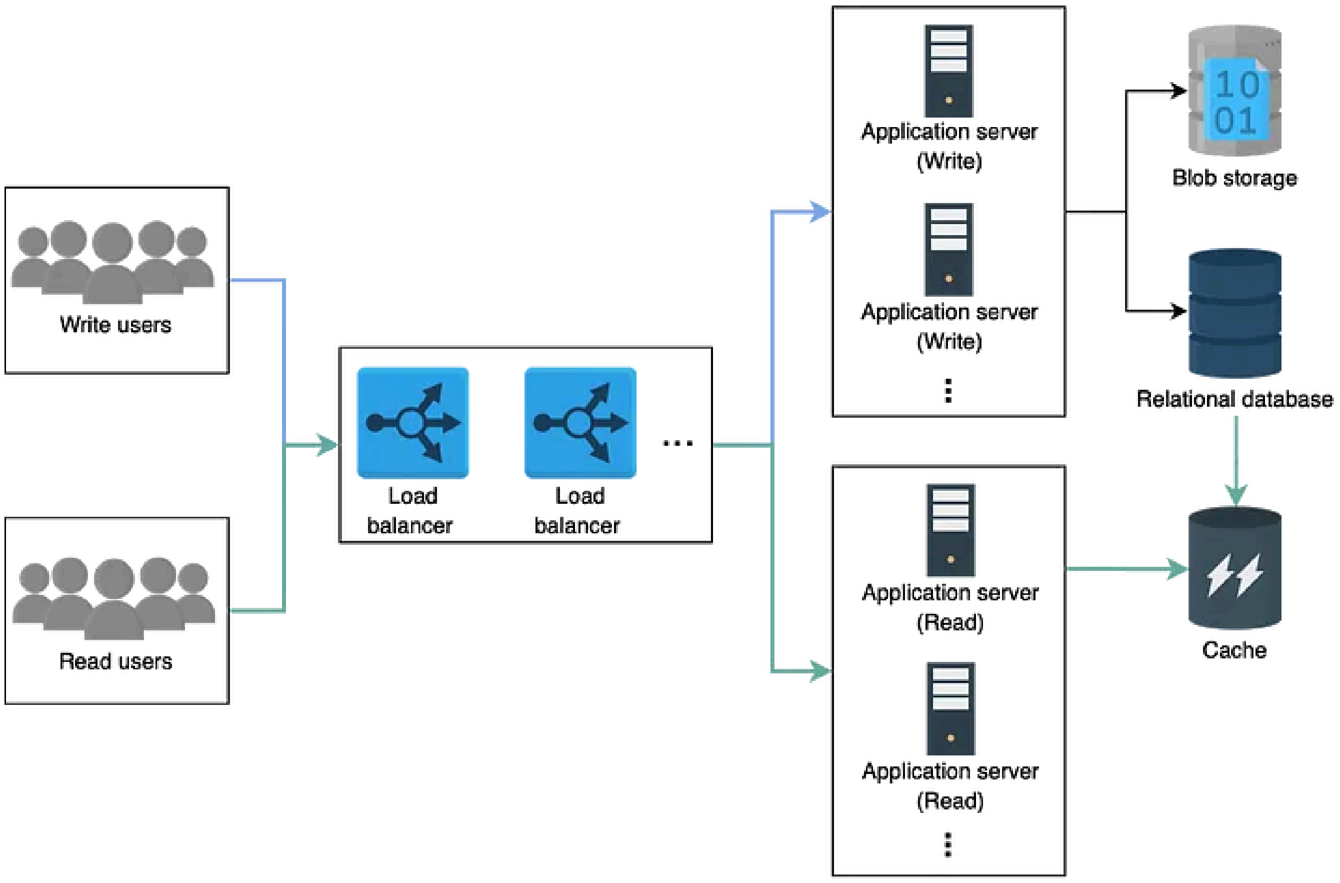

Write users
Read users
Load balancer
Load balancer
...
Application server (Write)
Application server (Write)
Application server (Read)
Application server (Read)
Blob storage
Relational database
Cache

Fig 3.0: Read/Write operations on photos

Generate a timeline

The pull approach — When a user opens their Instagram, we *send a request for timeline generation*. First, we fetch the list of people the user follows, get the photos they recently posted, store them in queues, and display them to the user. But this approach is *slow to respond* as we generate a timeline every time the user opens Instagram.

We can substantially reduce user-perceived latency by *generating the timeline offline*. For example, we *define a service that fetches the relevant data for the user before*, and as the person opens Instagram, it displays the timeline. This decreases the latency rate to show the timeline.

The push approach — In a ***push approach***, every user is responsible for pushing the content they posted to the people's timelines who are following them. In the previous approach, we pulled the post from each follower, but in the current approach, we push the post to each follower.

Now we only need to fetch the data that is pushed towards that particular user to generate the timeline.

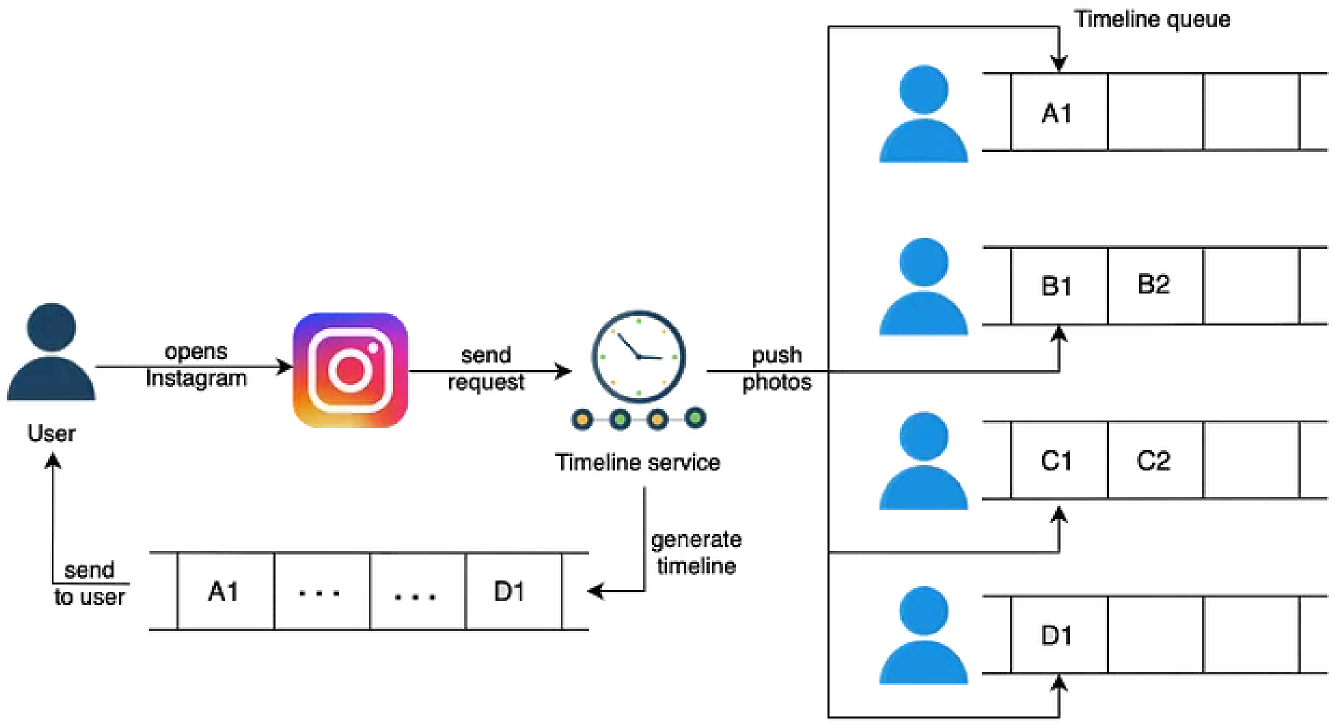


Fig 4.0: Push based approach

Hybrid approach — Let's split our users into two categories:

- **Push-based users:** The users who have a followers count of hundreds or thousands.
- **Pull-based users:** The users who are celebrities and have followers count of a hundred thousand or millions.

The timeline service pulls the data from pull-based followers and adds it to the user's timeline. The push-based users push their posts to the timeline service of their followers so the timeline service can add to the user's timeline.

Where do we store the timeline?

We store a user's timeline against a `userID` in a **key-value store**. Upon request, we fetch the data from the key-value store and show it to the user. The key is `userID`, while the value is timeline content (links to photos and videos). Because the storage size of the value is often limited to a few MegaBytes, we can store the timeline data in a blob and put the link to the blob in the value of the key as we approach the size limit.

Instagram story

We can add a new feature called story to our Instagram. In the story feature, the users can add a photo that stays available for others to view for 24 hours only. We can do this by **maintaining an option in the table** where we can **store a story's duration.** We can set it to 24 hours, and the **task scheduler** deletes the entries whose time exceeds the 24 hours limit.

Finalized design

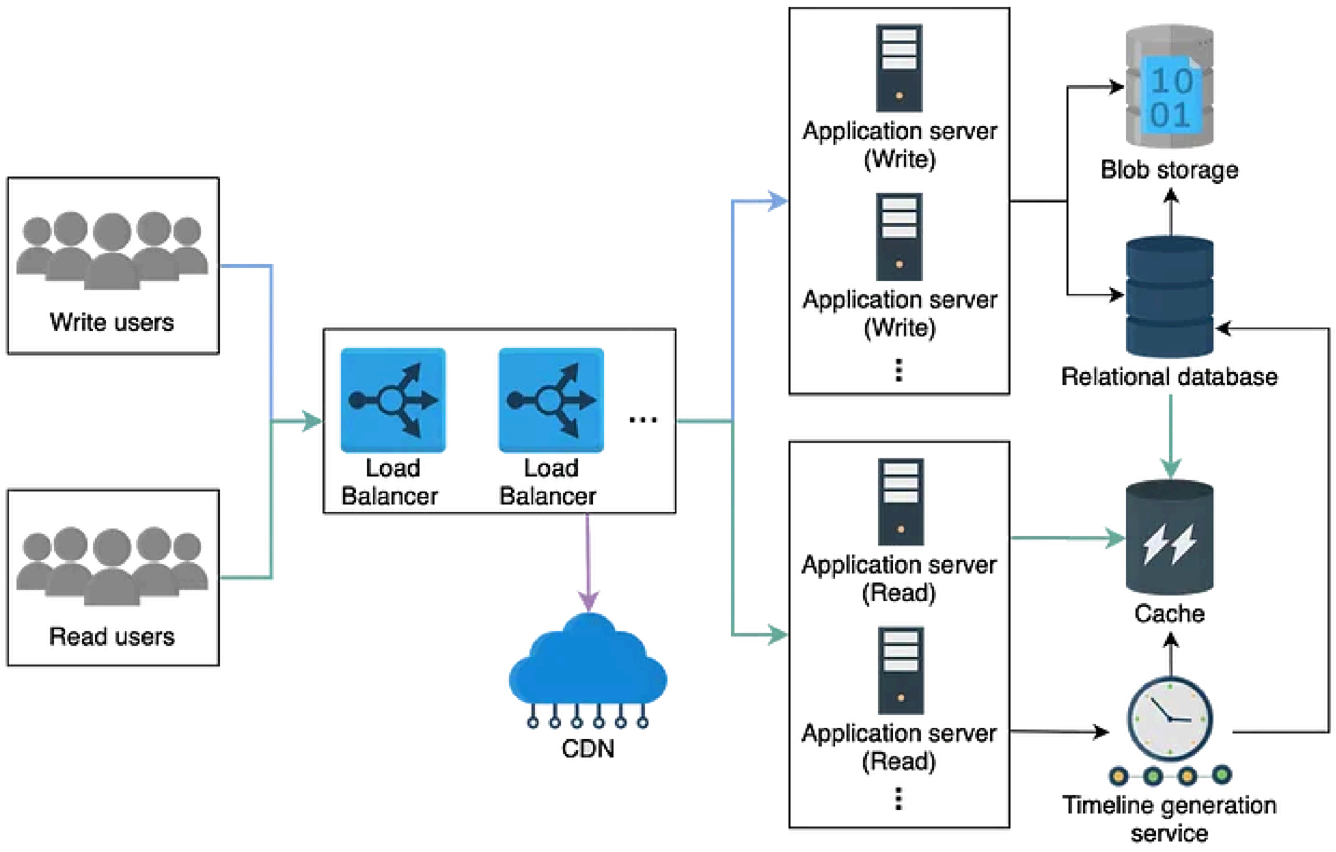

Write users
Read users
Load Balancer
Load Balancer
...
CDN
Application server (Write)
Application server (Write)
Application server (Read)
Application server (Read)
Blob storage
Relational database
Cache
Timeline generation service

Fig 5.0: The final design of Instagram

Evaluation

- **Scalability:** We can **add more servers** to application service layers to make the scalability better and handle numerous requests from the clients. We can also increase the number of databases to store the growing users' data.
- **Latency:** The use of cache and **CDNs** have reduced the content fetching time.

- **Availability:** We have made the system available to the users by using the storage and databases that are **replicated** across the globe.
- **Durability:** We have persistent storage that maintains the **backup** of the data so any uploaded content (photos and videos) never gets lost.
- **Consistency:** We have used storage like blob stores and databases to keep our data consistent globally.
- **Reliability:** Our databases handle **replication and redundancy,** so our system stays reliable and data is not lost. The load balancing layer routes requests around failed servers.

Instagram System Design Interview Software Architect Software Architecture Distributed Systems

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium